Flutter Architektur

Von einzelnen Files zu einer skalierbaren Architektur

Bevor es los geht

Vorstellung, Expertise

Wer bin ich?

Markus Kühle

Gründer & Geschäftsführer der coodoo GmbH,

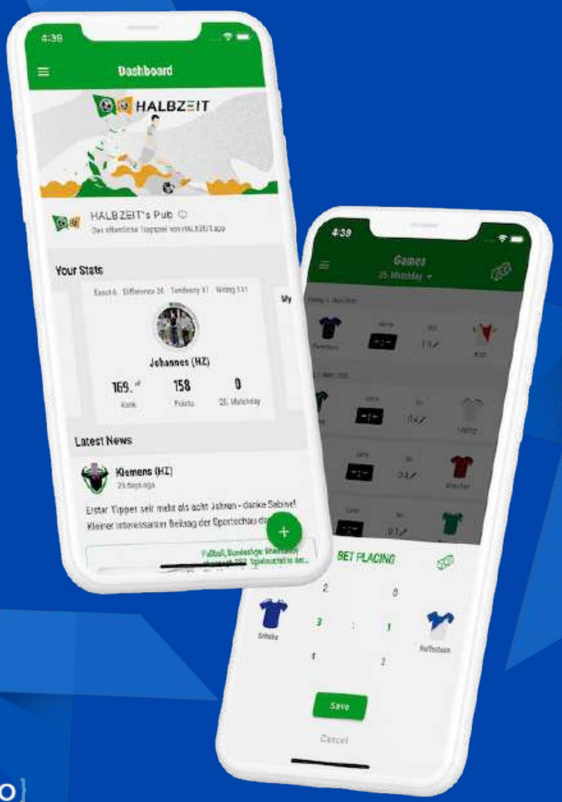
einer Enterprise Software Dienstleistungsfirma.

Seit 2019 Flutter Entwicklung für Halbzeit.app

und Kundenprojekte für Mobile Apps und Web.

@makueh https://coodoo.de

Halbzeit.app Expertise



- Öffentliches & White Label Fußball Tippspiel
- Mehrere 100k Spieler
- Über 50.000 Downloads
- Web, iOS, Android und Tablet
- AWS Cloud Service Integration
- Mandantenfähig mit vielen konfigurierbaren Details

https://halbzeit.app

{coodoo}

Warum Architektur

Relevanz, Projektgröße, Separation of Concerns

Relevanz einer Architektur

Eine gute Architektur zu wählen ist essentiell.

Strukturierter Code lässt sich besser warten, wenn die App wächst.

Keine Architektur

Keine Organisation des Codes
Endet bei größeren Apps in Chaos

Over Engineering

Bereits kleine Änderungen sind
nur sehr schwer umzusetzen

Wann über Architektur nachdenken?

- **Größe der App**
  Wie viele Pages, Navigationstiefe, Komplexität?

- **Größe des Teams**
  Wie viele Teammitglieder arbeiten gleichzeitig an der App?

- **Verteilung des Teams**
  Wie ist der direkte Austausch des Teams möglich?

- **Verständnis der Sprache und Struktur**
  Sprechen alle Mitglieder die gleiche “Flutter-Sprache”?

Separation of Concerns

UI

```dart
floatingActionButton: Row(
  mainAxisAlignment: MainAxisAlignment.end,
  children: [
    FloatingActionButton(
      onPressed: () =>
          ref.read(counterControllerProvider.notifier).decrement(),
      tooltip: 'Decrement',
      child: const Icon(Icons.remove),
    ), // FloatingActionButton
    const SizedBox(width: 16),
    FloatingActionButton(
      onPressed: () =>
          ref.read(counterControllerProvider.notifier).increment(),
      tooltip: 'Increment',
      child: const Icon(Icons.add),
    ), // FloatingActionButton
```

Business Logic

```dart
class CounterController extends StateNotifier<CounterState> {
  CounterController({required this.counterRepo})
      : super(const CounterState()) {
    init();
  }

  final CounterRepo counterRepo;

  void init() {
    state = state.copyWith(value: AsyncValue.data(counterRepo.value))
  }

  Future<void> increment() async {
    state = state.copyWith(value: const AsyncValue.loading());
    state = state.copyWith(
        value: await AsyncValue.guard(() => counterRepo.increment()))
  }
```

Flutter Vorgaben & Konventionen

Flutter gibt dem Entwickler **große Freiheiten** bei Architekturentscheidungen.

Flutter hat **keine Vorgabe** zu Ordnerstruktur oder anderen Konventionen bezüglich einer Architektur.

Konventionen & Abstimmungen

Ordnerstruktur, Dateinamen, Lint-Regeln

Ordner Konventionen

Layers first

```
‣ lib
  ‣ src
    ‣ presentation
      ‣ feature1
      ‣ feature2
    ‣ application
      ‣ feature1
      ‣ feature2
    ‣ domain
      ‣ feature1
      ‣ feature2
    ‣ data
      ‣ feature1
      ‣ feature2
```

Feature first

```
‣ lib
  ‣ src
    ‣ features
      ‣ feature1
        ‣ presentation
        ‣ domain
        ‣ repository
        ‣ entity
      ‣ feature2
        ‣ presentation
        ‣ domain
        ‣ repository
        ‣ entity
```

Es geht bei einem Feature nicht um die UI.
Domain-Driven Design als Startpunkt.

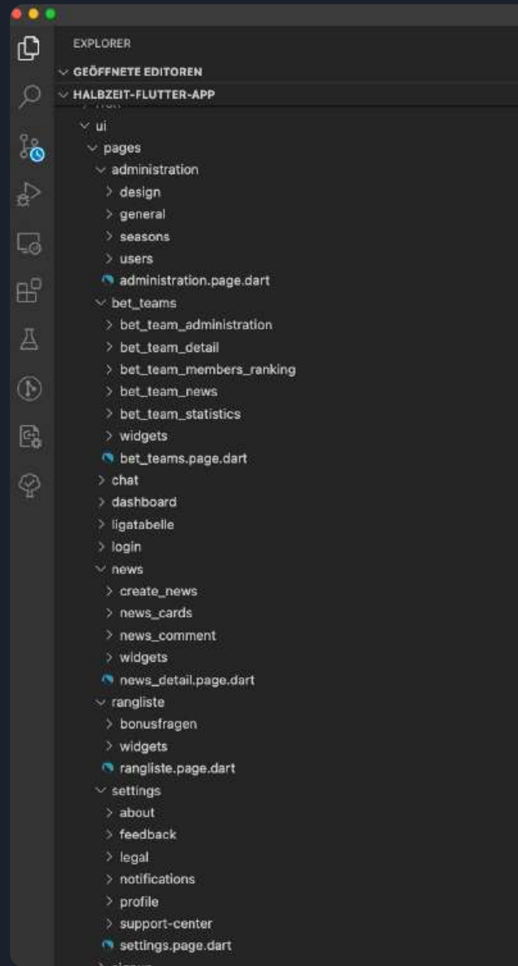
Layers first

Vorteile

- Schneller und einfacher Start
- Einfach zu verstehen

Nachteile

- Sehr unübersichtlich, sobald die App wächst
- Zusammengehörende Files für ein Feature über das Projekt verteilt



Beispiel Halbzeit.app Flutter App erste Version

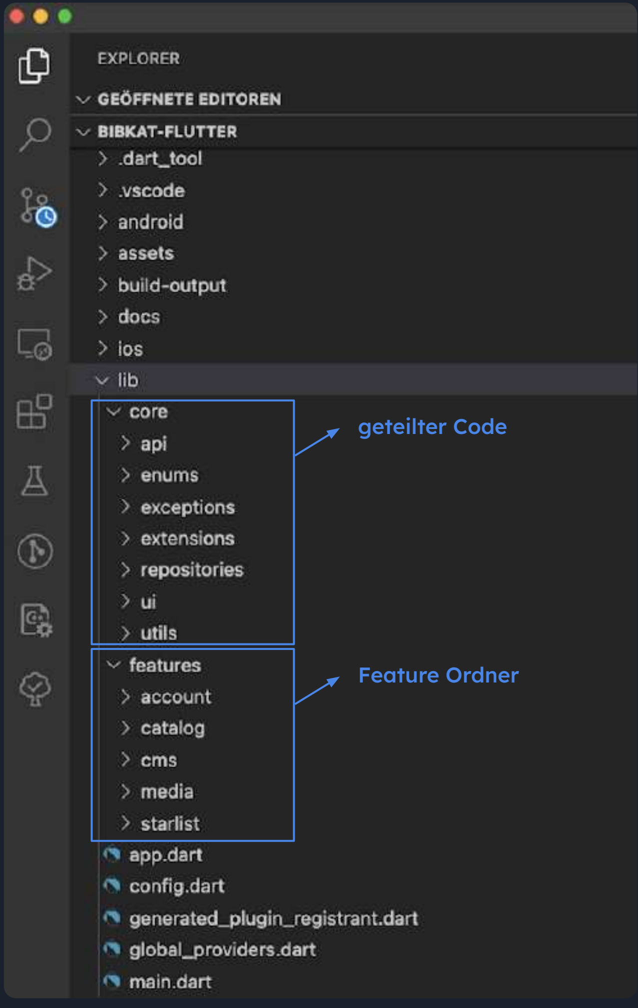
Feature first

Vorteile

- Files, die zu einem Feature gehören, sind an einem Ort zu finden
- Layers in Feature
- Einfache Kommunikation

Nachteile

- Verständnis, was ein Feature ist, muss im Team trainiert werden



{coodoo}

Feature first - Startpunkt

Es geht bei einem Feature **nicht** um die UI.

Domain-Driven Design als **Startpunkt**.

Domain-Driven Design ist ein Ansatz für die Softwareentwicklung, der die Entwicklung auf die Programmierung eines Domänenmodells konzentriert, das über ein umfassendes Verständnis der Prozesse und Regeln einer Domäne verfügt.

Martin Fowler über das Buch von Eric Evans von 2003

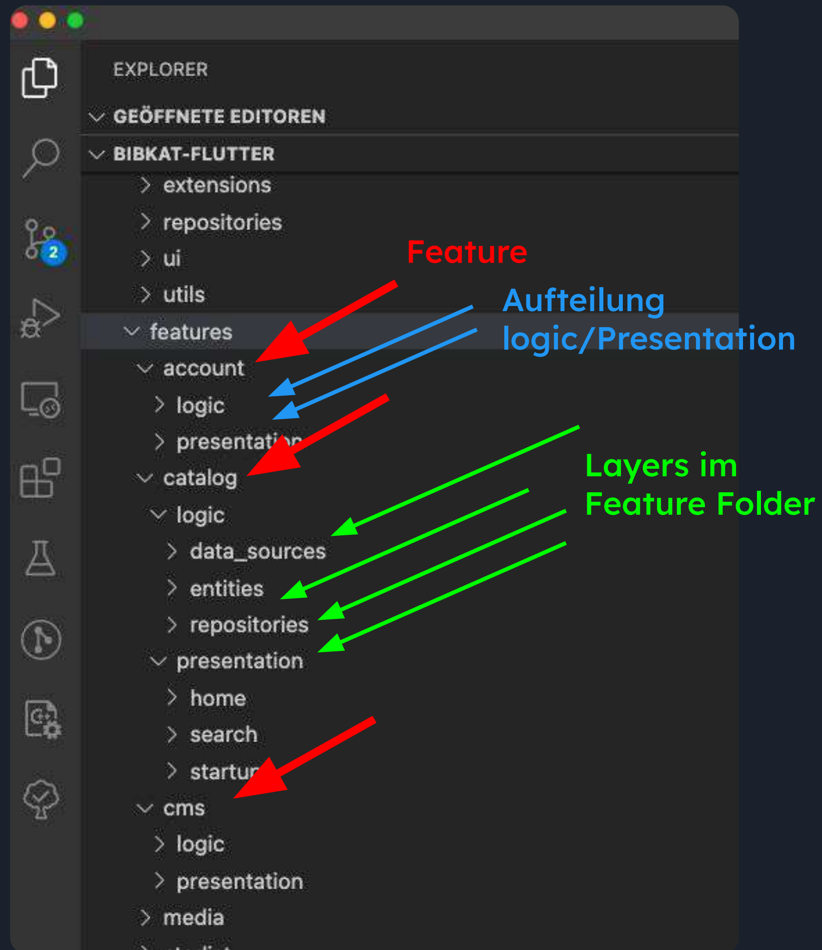
https://martinfowler.com/bliki/DomainDrivenDesign.html

Feature first

Layers im Feature

Die Layer werden innerhalb des Features als Ordner eingerichtet.

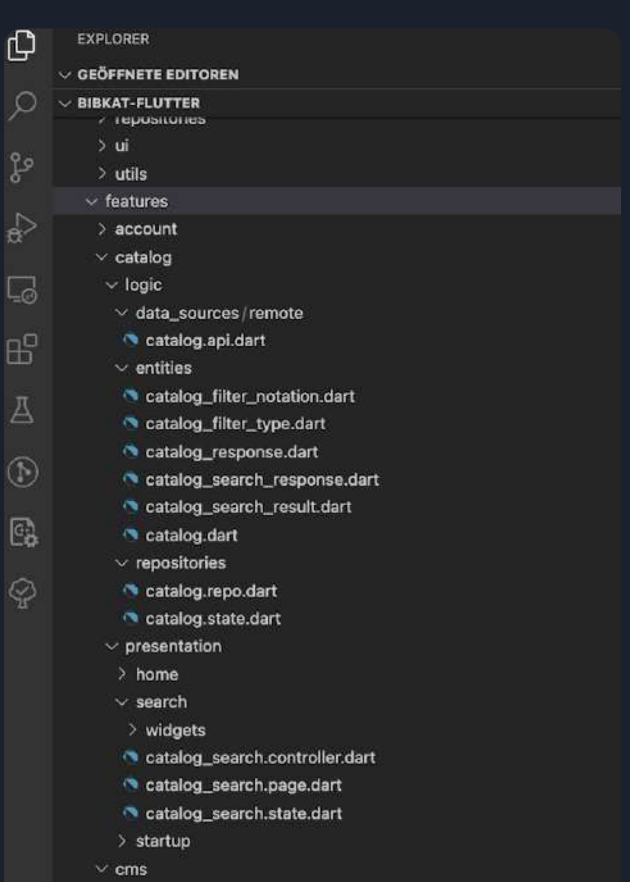
Wir unterteilen noch einmal in `logic` und `presentation`.



Konventionen für Dart Files & Klassen

- Filename sagt über **Art des Inhalts**
  aus Bsp.: `catalog_search.page.dart`, `catalog.repo.dart`,

- **Eine** Klasse pro File
  Ausnahme sind eng zusammengehörige Klassen wie z.B. Stateful Widgets

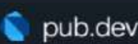


{coodoo}

Flutter Lint Regeln

- Linter im Default Projekt bereits konfiguriert
- Flutter Lints basiert auf den empfohlenen Dart Lints
- In `analysis_options.yaml` anpassbar
- Aufrufen mit `flutter analyze`

Alle konfigurierten Lint Regeln immer beachten!



Published 32 days ago · flutter.dev (Null safety)

pub v2.0.1

This package contains a recommended set of lints for Flutter apps, packages, and plugins to encourage good coding practices.

This package is built on top of Dart's `recommended.yaml` set of lints from package:lints.

Lints are surfaced by the dart analyzer, which statically checks dart code. Dart-enabled IDEs typically present the issues identified by the analyzer in their UI. Alternatively, the analyzer can be invoked manually by running `flutter analyze`.

Usage

Flutter apps, packages, and plugins created with `flutter create` starting with Flutter version 2.3.0 are already set up to use the lints defined in this package. Entities created before that version can use these lints by following these instructions:

1. Depend on this package as a **dev_dependency** by running `flutter pub add --dev flutter_lints`.
2. Create an `analysis_options.yaml` file at the root of the package (alongside the `pubspec.yaml` file) and `include: package:flutter_lints/flutter.yaml` from it.

Example `analysis_options.yaml` file:

https://pub.dev/packages/flutter_lints

Lint Beispiel - forEach()

```
Run | Debug | Profile
3   void main(  void forEach(void Function(String) action)
4     runApp(c
5   }           Type: void Function(void Function(String)) dart:core
6
7   class MyAp  Invokes [action] on each element of this iterable in iteration order.
8     const My
                Example:
9
                final numbers = <int>[1, 2, 6, 7];
10    // This   numbers.forEach(print);
11    @overrid  // 1
12    Widget b  // 2
13              // 6
14      List<S  // 7
15      name
16      names.  Avoid using `forEach` with a function
17      names.  literal. dart(avoid_function_literals_in_foreach_calls)
18      names.forEach((element) => print(element));
19
```

Lint Github Regel

Effective Dart Usage

Mögliche Architekturen

MVC, MVC+S, Clean Architecture, State Architecture

Übersicht der Architekturen

- Es existieren eine Menge an Architekturen
- MVC, MVC+S, MVP, MVVM, Clean Architecture, Android App Architecture…
- Flutter gibt keine Architektur vor

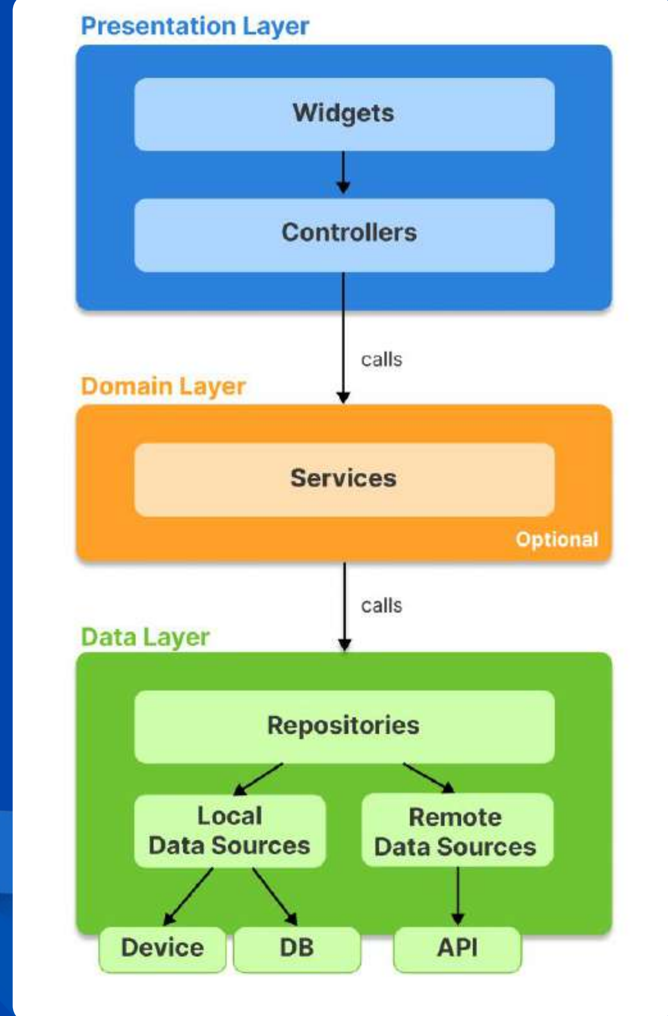
Populäre State Architekturen

- **Bloc Architektur** (basierend auf der Bloc Library - bloclibrary.dev)
- **Riverpod Architektur** (MVVM State - riverpod Package - pub.dev/packages/riverpod)

Die Architekturen basieren auf Provider, welches das von Flutter empfohlene State Management Package ist.

State Architektur

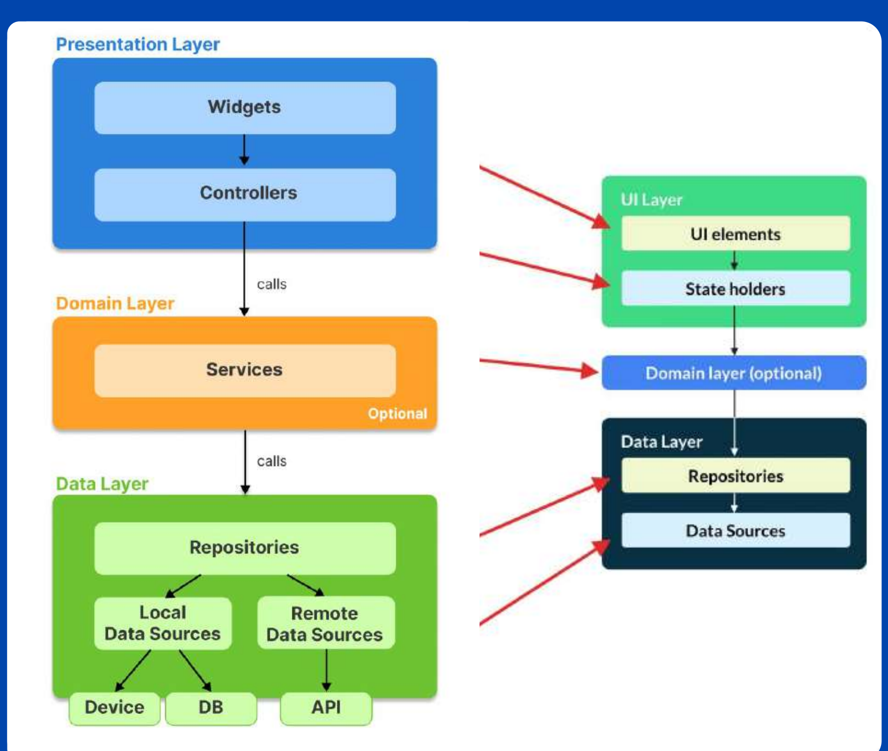
- Basierend auf State Management
- Drei Layer:
  - Presentation
  - Domain
  - Data
- Jeder Layer hat eine bestimmte Aufgabe
- Klarer Vertrag wie die Kommunikation stattfindet.



Android Architecture

- State Architektur
- 3 Layer
- Kommt unserer Zielarchitektur am nächsten

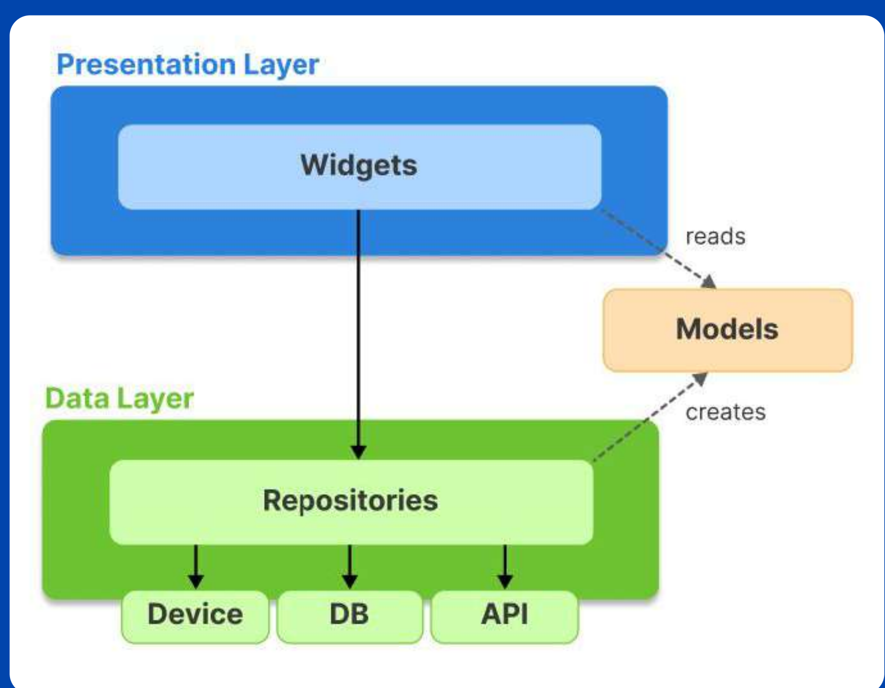
Android Guide to app architecture



{coodoo}

Minimale State Architektur

- Kleinste mögliche Architektur
- Zwei Layer:
  - Data
  - Presentation
- Kernbestandteile:
  - State Management
  - Dependency Injection
- Direkt mit Entities anstatt DTOs arbeiten



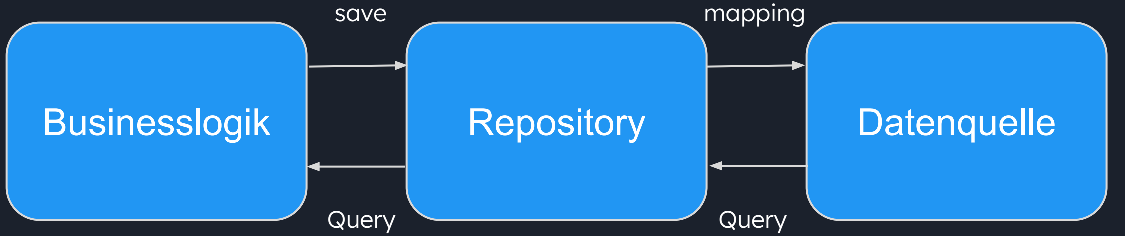
{coodoo}

Data Layer - Repository

Repository Pattern, Entities, Final & Immutable, Freezed

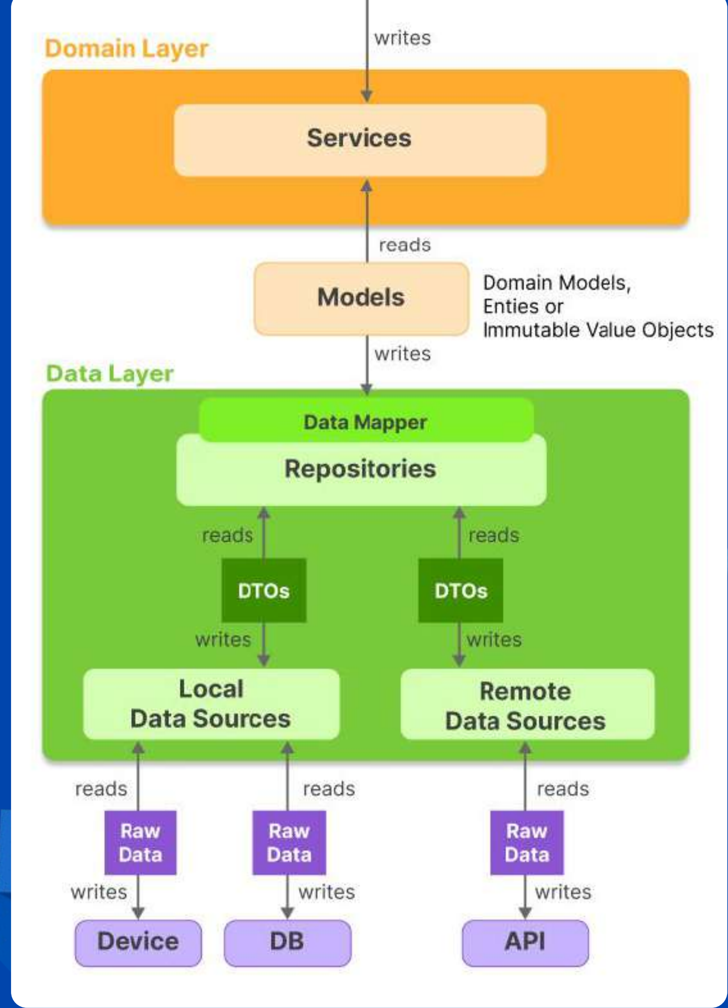
Repository Pattern

- Auftrennung zwischen Businesslogik und Datenbeschaffung
- Zentrale Zuständigkeit, um Daten zu beschaffen
- Ist unabhängig von der Datenquelle
- Lädt Daten und mappt auf Objekte
- Repository lädt selbst nicht die Daten, sondern hilft bei der Datenbeschaffung
- Hilft bei den Tests, denn es lässt sich schnell austauschen



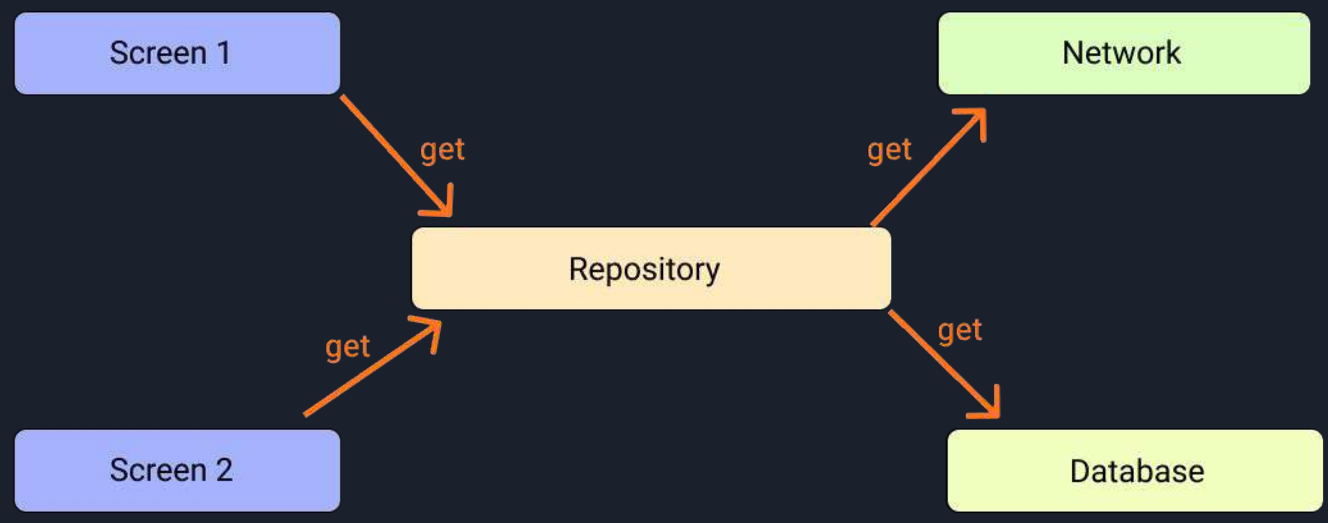
Repository Pattern in der Architektur

- Layer nennt man auch Infrastructure
- Unterscheidung zwischen Local und Remote Data Sources
- Repository managed nur den Aufruf der Data Source
- Wandelt DTOs in Enties um



{coodoo}

Repository Struktur



Dartpad

Model (Entities)

The entities must be our data types or classes that are used in different parts of our software.
Uncle Bob

- Entities beschreiben das Domain Model
- Werden in der Domain Layer erstellt und verwaltet
- Domain Layer wendet über Business Logik CRUD Operationen auf Entities an.
- In der kleinsten Architektur können sie direkt im Repository erstellt und nach oben gereicht werden.

Immutable Objekte & Values

A class is immutable if all of the instance fields of the class, whether defined directly or inherited, are final.
API Documentation

- Immutable Objekte können ihren Status nach der Erstellung nicht mehr ändern.
- Vorteile in der Performance
- Nur eine Stelle/Layer kann/darf Entities verändern und den State der Anwendung ändern.
- In Dart wird mit `@immutable` auf Klassenebene angezeigt, dass diese Klasse nicht veränderbar ist (https://api.flutter.dev/flutter/meta/immutable-constant.html)

Immutable Object

Wir arbeiten mit unveränderbaren Objekten.

Objekt enthält nur final Attribute.

`UnmodifiableListView` ermöglicht hier eine nicht veränderbare Liste.

Empfänger (Consumer) der Liste sollen die Daten nicht verändern können.

Alternative: freezed Package.

```
@immutable
class Birthday {
 final String name;
 final DateTime date;
 final String? profileImage;
 final String? notes;

 const Birthday({required this.name, required
this.date, this.profileImage, this.notes});

  …
}
```

```
UnmodifiableListView<Birthday> get birthdays =>
UnmodifiableListView(_birthdays);
```

{coodoo}

Freezed Package

Generiert den notwendigen Code einer unveränderlichen Klasse:

- Konstruktoren
- toString, equals & hashCode Methode
- copyWith Methode
- JSON Serialisierung

https://pub.dev/packages/freezed

```dart
@immutable
class Person {
  const Person({
    required this.firstName,
    required this.lastName,
    required this.age,
  });

  factory Person.fromJson(Map<String, Object?> json) {
    return Person(
      firstName: json['firstName'] as String,
      lastName: json['lastName'] as String,
      age: json['age'] as int,
    );
  }

  final String firstName;
  final String lastName;
  final int age;

  Person copyWith({
    String? firstName,
    String? lastName,
    int? age,
  }) {
    return Person(
      firstName: firstName,
      lastName: lastName,
      age: age,
    );
  }

  Map<String, Object?> toJson() {
    return {
      'firstName': firstName,
      'lastName': lastName,
      'age': age,
    };
  }

  @overrride
  String toString() {
    return 'Person('
      'firstName: $firstName, '
      'lastName: $lastName, '
      'age: $age'
    ')';
  }

  @override
  bool operator ==(Object other) {
    return other is Person &&
        person.runtimeType == runtimeType &&
        person.firstName == firstName &&
        person.lastName == lastName &&
        person.age == age;
  }

  @override
  int get hashCode {
    return Object.hash(
      runtimeType,
      firstName,
      lastName,
      age,
    );
  }
}
```

```dart
@freezed
class Person with _$Person {
  const factory Person({
    required String firstName,
    required String lastName,
    required int age,
  }) = _Person;

  factory Person.fromJson(Map<String, Object?> json)
      => _$PersonFromJson(json);
}
```

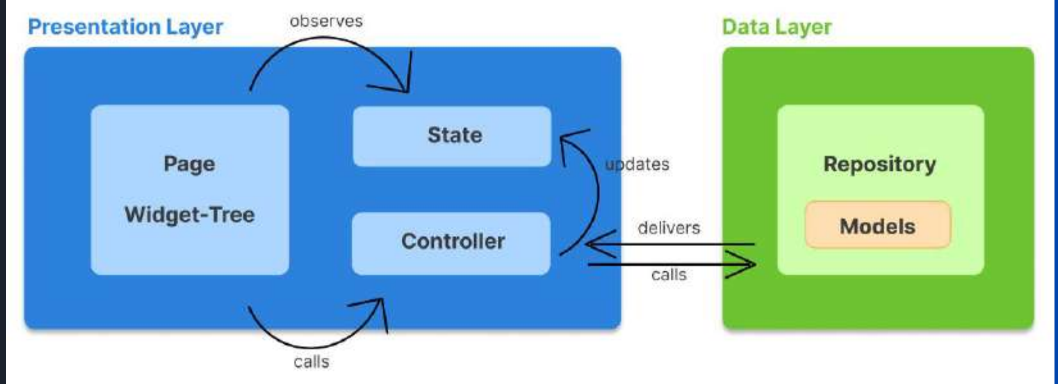
Generieren lassen durch https://app.quicktype.io

Presentation Layer - States

Repository Pattern, Entities, Final & Immutable, Freezed

Datenfluss Presentation Layer

- Page mit Widgets zeigt Daten aus dem State
- State wird durch den Controller aktualisiert
- Page ruft Aktionen des Controllers auf
- Controller lädt aus Repository Daten und Updated State

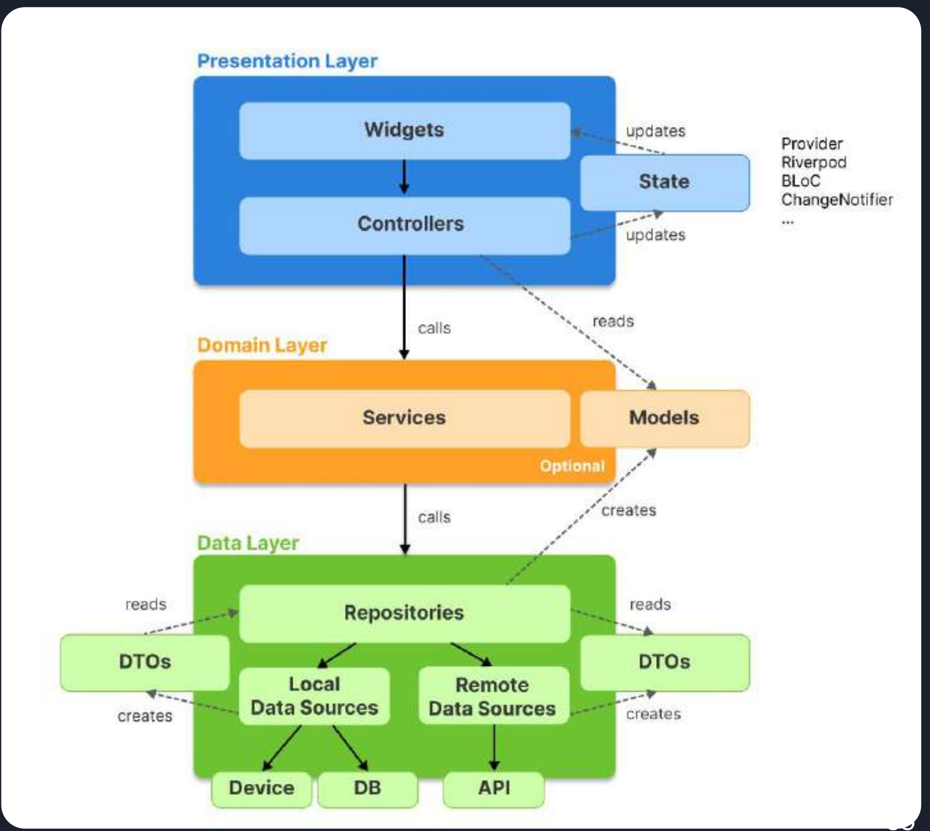


{coodoo}

State Management Einführung

State Management in der Architektur

- Einordnung des State Management in eine möglichen Architektur
- Presentation Logic Holders
- Daten werden in Repositorys oder in der Domain Schicht geändert und manipuliert -> die View muss darauf reagieren



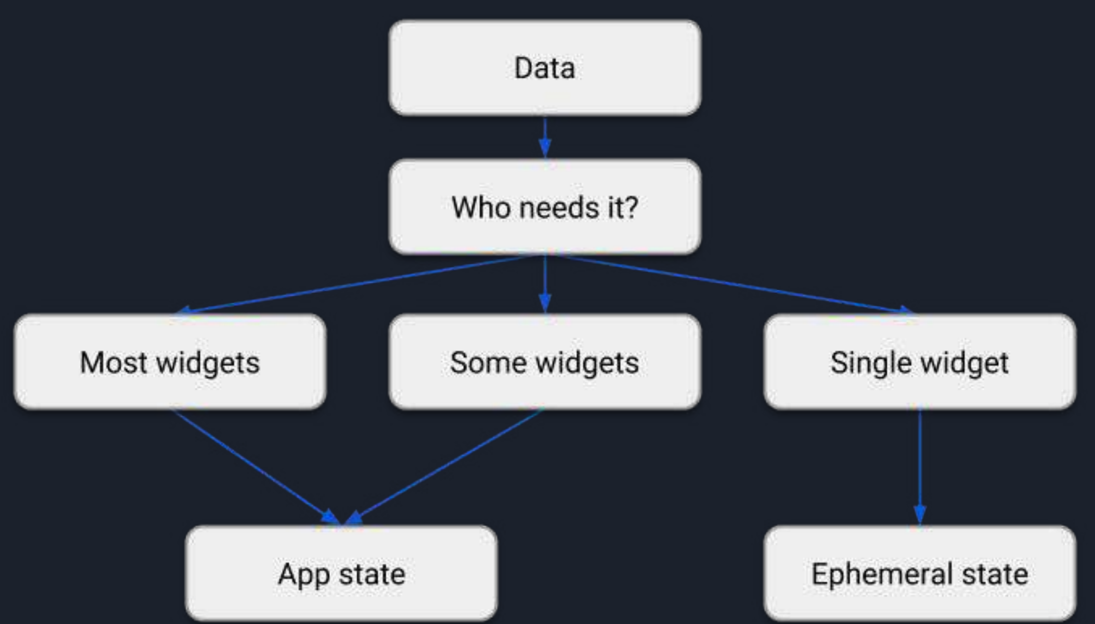
Was ist ein State?

- Flutter ist deklarativ
  -> Die UI wird gebaut, um den **aktuellen State** der App darzustellen.

- Wenn der State sich ändert, wird die UI vollständig neu gerendert.

- Unterscheidung zwischen **Ephemeral state** (flüchtig) und **App state** (übergreifend)

Ephemeral state & App state



Keine ganz klare Regel wann eine Variable zu dem Ephemeral State oder App State gehört.

Problem

Zentrale Daten sollen auf einem Widget weiter unten im Widget-Tree angezeigt werden.

Bisherige einzige Lösung:
Die Daten über den Constructor nach unten durchreichen.

State Management Lösungen

Stateful Widgets	(Flutter build in)
InheritedWidget & InheritedModel	(Flutter build in)
Provider (von Flutter empfohlen)	pub.dev https://pub.dev/packages/provider
Riverpod	pub.dev https://pub.dev/packages/flutter_riverpod
BLoC / Rx	pub.dev https://pub.dev/packages/flutter_bloc
Redux	pub.dev https://pub.dev/packages/flutter_redux

und noch einige weitere..

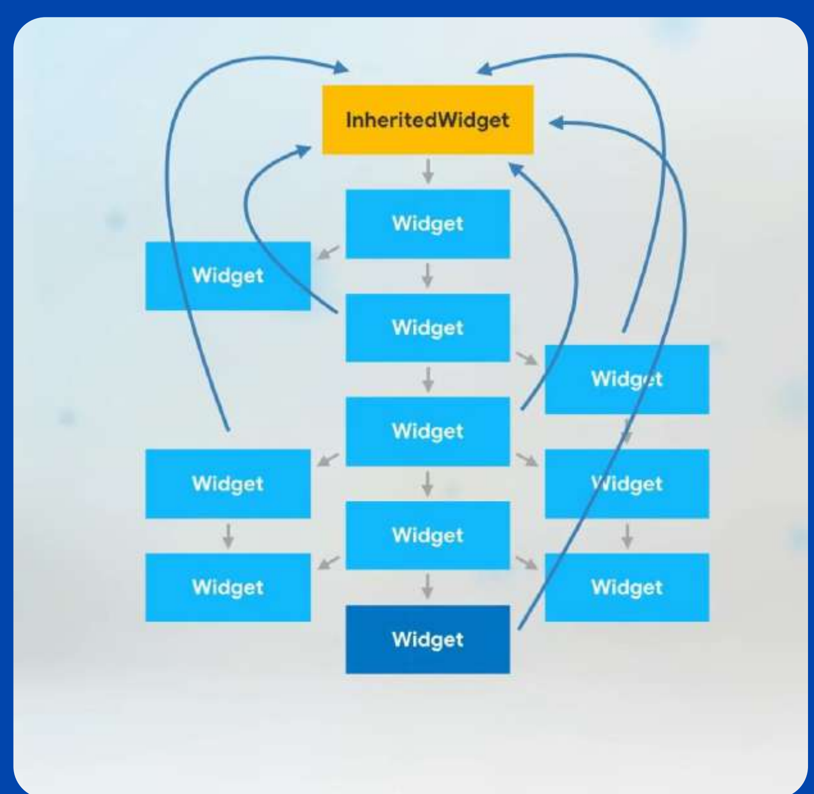
https://docs.flutter.dev/development/data-and-backend/state-mgmt/options
https://twitter.com/RydMike/status/1528827017172504579

InheritedWidget Funktion

- Direkter Zugriff auf das Inherited Widget
- Es müssen keine Daten durchgegeben werden.

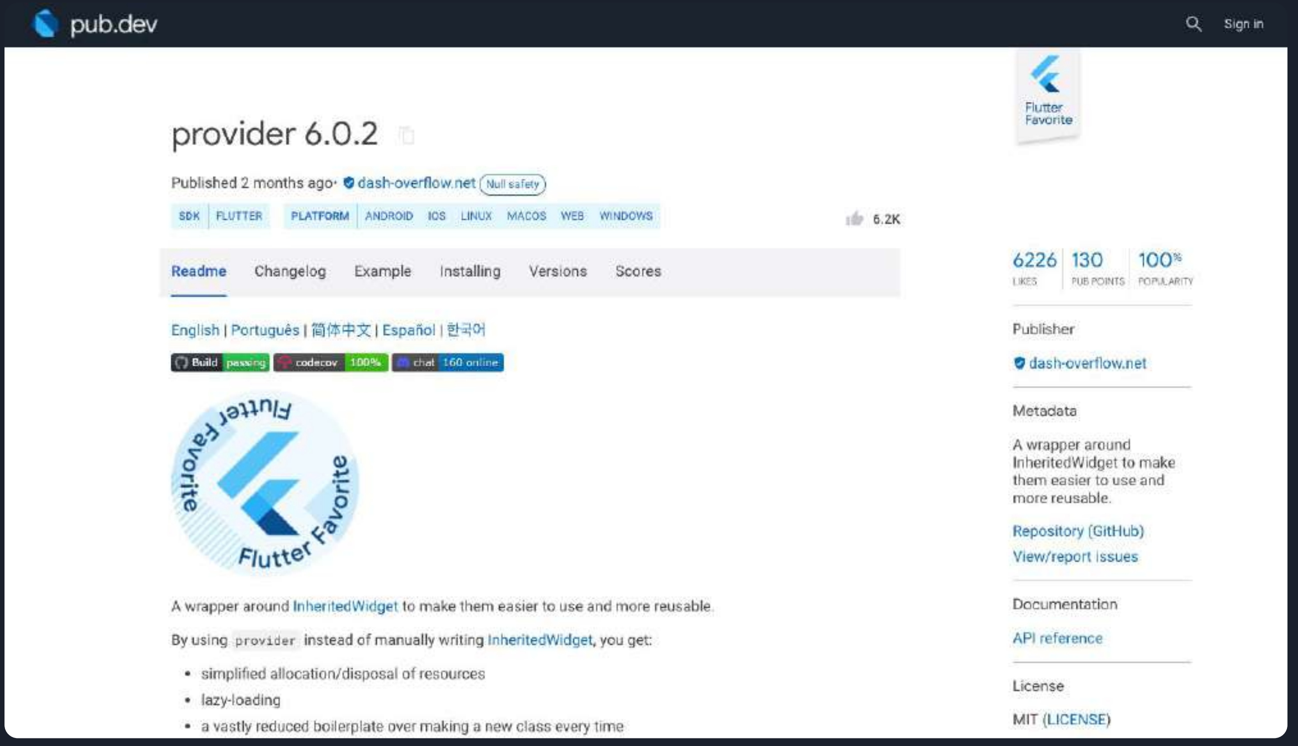
Aus dem Counter Beispiel:

```
children: <Widget>[
  const Text(
    'You have pushed the button this many times:',
  ), // Text
  Text(
    '$_counter',
    style: Theme.of(context).textTheme.headline4,
  ), // Text
], // <Widget>[]
```



State Management mit Provider

Provider Package



https://pub.dev/packages/provider

{coodoo}

Was bekommt man mit Provider?

“A wrapper around `InheritedWidget` to make them easier to use and more reusable.”

Einen bequemen Wrapper, um das `InheritedWidget` sehr einfach verwenden zu können.

ChangeNotifier

Ein Notifier gibt bekannt, wenn sich etwas geändert hat.

Alternative für nur ein Value ist der `ValueNotifier`.

```
class BirthdayRepo extends ChangeNotifier {
 static final BirthdayRepo _birthdayRepo =
BirthdayRepo._internal();

 factory BirthdayRepo() {
   return _birthdayRepo;
 }

 BirthdayRepo._internal();

 final List<Birthday> _birthdays = [];

 UnmodifiableListView<Birthday> get birthdays =>
     UnmodifiableListView(_birthdays);

  Birthday insert(Birthday birthday) {
   _birthdays.add(birthday);
   notifyListeners();
   return birthday;
  }
  ......
}
```

Provider

Managed den Lifecycle konfigurierten Notifier (Models)

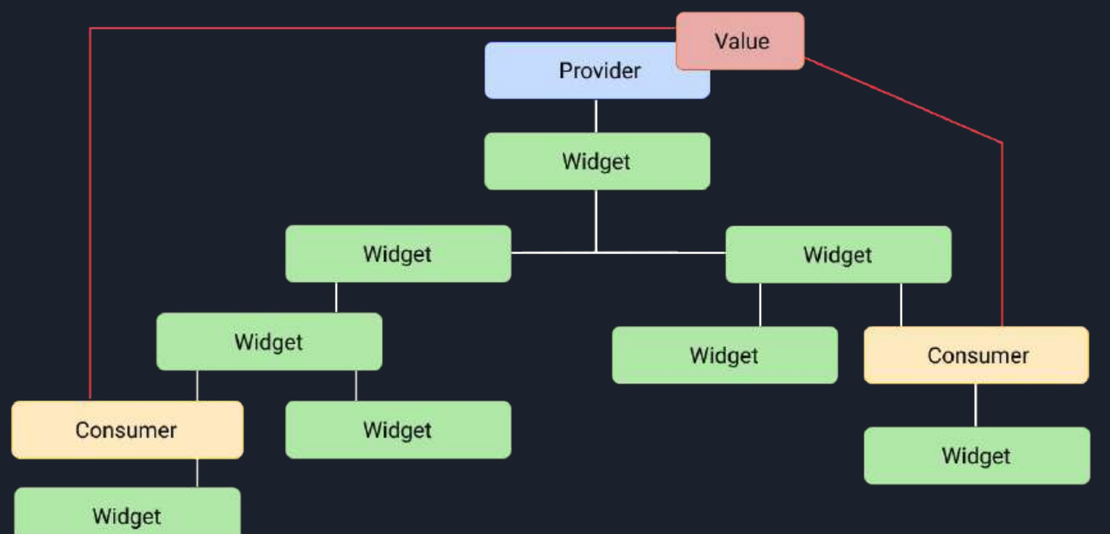
Typische Provider

- MultiProvider
  Wird viel verwendet, weil es ein einfacher und schneller Weg ist, verschiedenste Provider zu initialisieren.

- ChangeNotifierProvider
  Kann verwendet werden, wenn ein ChangeNotfier genutzt wird.

```
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(
            create: (_) => BirthdayRepo()),
      ],
      child: const App(),
    ),
  );
}
```

Consumer



Ein `Consumer` Widget hört auf die Änderung des `Value` und zeichnet anschließend sich selbst und die Widgets unterhalb erneut.

`Consumer` ist Teil des Provider Package.

Consumer API-Link

Consumer Widget

Ein Consumer Widget hört auf Änderungen des States.

`context.watch()`: Widget erhält State und hört auf Veränderung

`context.read()`: Widget erhält State und ignoriert Änderungen.

```
class BirthdaysScreen extends StatelessWidget {
 const BirthdaysScreen({Key? key}) : super(key:
key);

 @override
 Widget build(BuildContext context) {
   final birthdays =
context.watch<BirthdayRepo>().birthdays;
   …

           onDismissed: (direction) {

context.read<BirthdayRepo>().delete(birthday);
             …
             );
           },

}
```

{coodoo}

State Management mit Riverpod

Was ist Riverpod?

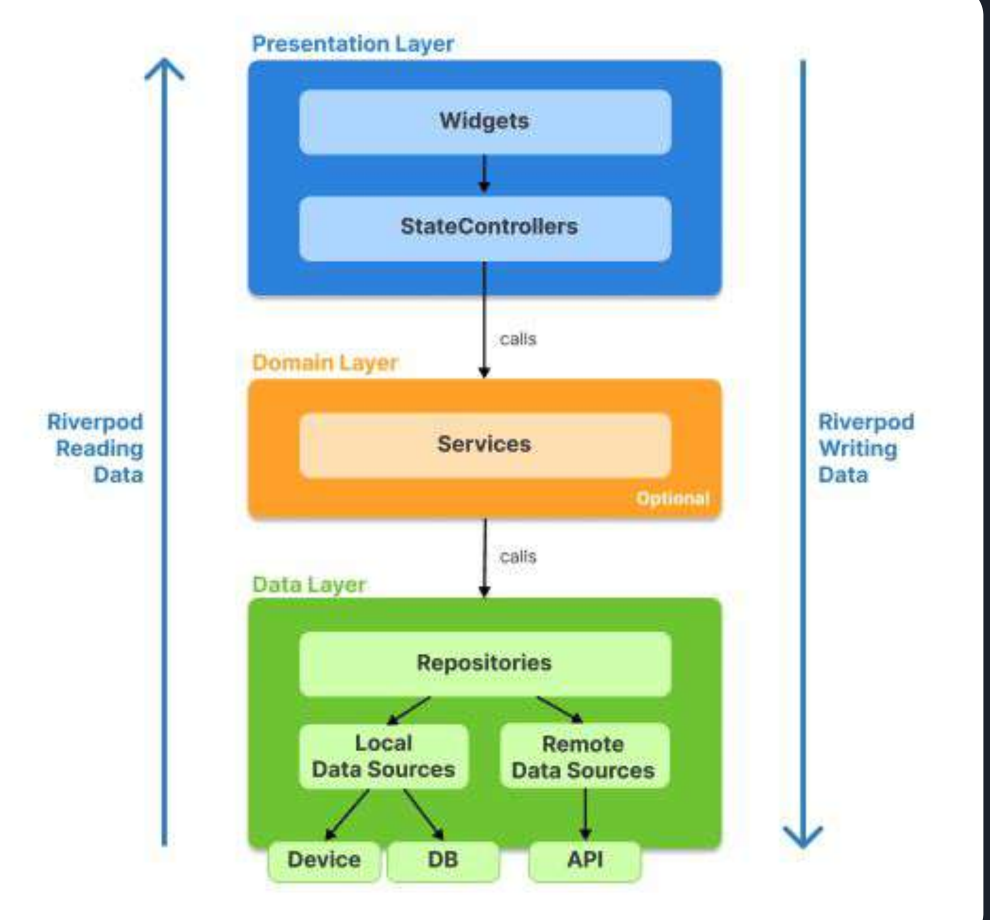
- Verwendet die Vorteile von Provider und bringt neue Erweiterungen mit
- Ermöglicht einfach Provider zu erstellen, zuzugreifen und zu kombinieren
- Hinterlässt Code, der einfach zu testen ist
- Zeigt Fehler bereits zur Compilezeit

This project can be considered as a rewrite of provider to make improvements that would be otherwise impossible.

https://riverpod.dev

Architektur und Datenfluss

Daten über die verschiedenen Schichten hinweg lesen und schreiben unter Verwendung von Riverpod.



{coodoo}

Was bietet Riverpod?

Riverpod **Provider sind global** und leben außerhalb des Widget Trees.

Verschiedene Provider für den jeweiligen Anwendungsfall.

Der Kern von Riverpod ist **Dependency Injection**.

Bietet State-Management, wie wir es von Provider kennen.

Beschreibt sich selbst als “**A Reactive Caching and Data-binding Framework**”

Provider Modifiers

Standard
Globaler Provider, der als Singleton agiert - kein Modifier.

```
final counterRepoProvider = Provider<int>((ref) => 8);
```

Family
Einen eindeutigen Provider anhand einer ID erhalten - Family Modifier.

```
final counterRepoProvider = Provider.family<int, String>((_, ref) { return 8; });
```

AutoDispose
Die Provider Instanz aus dem Speicher entfernen, wenn sie nicht mehr benötigt wird - AutoDispose.

```
final counterRepoProvider = Provider.autoDispose<int>((ref) => 8);
```

Über `ref` mit den Providern interagieren

ref.watch() Lädt die Daten des Providers und hört auf Änderungen (**observe**). Ändern sich die Daten wird das Widget neu gebaut (**rebuild**).
Verwendung: z.B. reaktiver Controller

ref.listen() Fügt einen Listener hinzu und führt eine bestimmte Aktion (**Callback**) aus, wenn sich die Daten im Provider ändern.
Verwendung: z.B. Snackbar anzeigen

ref.read() **Liest** den Daten eines Providers **nur einmal** und reagiert nicht auf Veränderung der Daten im Provider.
Verwendung: Standard Dependency Injection

Wenn immer möglich lieber ref.watch() anstatt ref.read() oder refl.listen() verwenden. Wenn man die App so implementiert, dass sie sich auf ref.watch() verlässt wird sie wartungsfreundlicher.

AsyncValue durch FutureProvider

Mit AsyncValue<T> wird garantiert, dass der Entwickler nicht vergisst auf die Loading & Error States zu reagieren.

FutureProvider

```
ref.watch(counterRepoProvider).when(
  data: (response) => Text("${counterRepoProvider.toString()}: $response"),
  loading: ()=> const CircularProgressIndicator(),
  error: (err, st) => Text("ERROR: ${err.toString()}")
)
```

FutureStates

Live Code Beispiel
https://github.com/coodoo-io/coodoo_style_flutter_project

Wichtigste Punkte

- Nicht jedes Projekt hat die gleiche Architektur
- Im Team eine einheitliche Projektsprache sprechen
- Das Team auf Konventionen und Strukturen trainieren
- State Architektur einführen und Team schulen
- Je größer das Projekt wird umso strikter die Architektur anwenden

Vielen Dank

markus-kuehle

https://coodoo.de

https://flutter.de